Testing Autonomous Systems for Deep Space Exploration

Kirk Reinholtz and Keyur Patel
Jet Propulsion Laboratory*
California Institute of Technology
4800 Oak Grove Drive MS 303-310
Pasadena, CA 91109 USA
first.last@jpl.nasa.gov

October 10, 1997
(revised December 2, 1997 Version 1.12)

ABSTRACT

NASA is moving into an era of increasing spacecraft autonomy. However, before autonomy Cal] be routinely utilized, we must provide techniques for providing assurance (list the system will perform correctly in flight. We describe why autonomous systems require advanced verification techniques, and offer some management and technical techniques for addressing the differences.

Autonomous ~oal-drive.n spacecraft require advances in verification techniques because optimization (e.g. planning and scheduling) algorithms are at the core of much of autonomy. It is the nature of such algorithms that over much of the input space an intuitively "small" change in the input results in a correspondingly "small" change in the output: This type of response typically leads one to conclude, quite reasonably, that if the two responses are correct, those responses "between" them will probably be correct. However, there are certain regions in the input space where a "small" change in the input will result in a radically different output: One is not so inclined to conclude that all responses in these transition zones are likely to be correct.

We believe, for two reasons, that these transition zones are one place where autonomous systems are likely to fail. First, boundary conditions, often a rich source of faults, arc highly exercised in the transition zones, and so increase the likelihood of faults. Second, within the transition zone the algorithm outputs arc likely to appear unusual, and, since the *outputs* of the algorithm become *inputs* to the remainder of the system, the whole system is probably pushed outside of its nominal usage profile: historically shown to be another good source of faults.

We close with a discussion of risk management. Autonomous systems have many well-kll(wll management risk factors. Risk management and quality concerns must be pervasive, throughout all team members and the whole life-cycle of the project.

*The work described was performed at the Jet Propulsion Laboratory, California Institute of Technology under contract with the National Aeronautics and Space Administration. Submitted to IEEE Aerospace Conference Proceedings(paper 035), Aspen, CO, March 21-28, 1998.

TABLE OF CONTENTS

1 INTRODUCTION

NASA [1] and other agencies [2] are moving into an era of increased spacecraft autonomy a natural outcome of a desire to reduce the cost of science data combined with the impact of light-time communication delays and the availability of ever more powerful computers. Autonomy has the potential to decrease the cost of spacecraft operations, improve reliability, and provide increased science product volume and quality. However, before these things can occur, we must provide a compelling argument that we earl deliver a flight-quality product.

Traditional spacecraft flight software testing at JPL basically demonstrates that each command works correctly, that combinations of commands that are likely to be used together during the mission work properly together, and that all interfaces are operating correctly. This has been appropriate and effective, because the spacecraft systems have been designed to minimize the influence of environmental factors 011 the execution of low-level commands.

However, almost by definition, as the degree of autonomy increases, the sensitivity to the environment also increases. Since the system is sensitive to the environment, and the actual mission environment can't be predicted with sufficient accuracy, one must explore the behavior of the system over a range of plausible environments in order to demonstrate the robustness of the system.

We propose a four-pronged mitigation plan. First, formal specifications of the correct behavior of the system must be developed, along with tools to validate an executing system against its specification, so that even minor departures can be detected. Otherwise, it is likely that "minor" divergences will not be detected until they become major divergences, perhaps during the mission.

Next, locate the transition zones, via a combination of analysis and search, then explore each in detail for incorrect behavior. Finally, manage the risk over the whole life-cycle.

A formal specification, against which an execution of the system can be verified in a white-box manner, is crucial. There are a great many details to be verified, many of which will be neglected if the verification is done manually. White-box testing increases test efficiency, because even faults that don't manifest themselves as divergent output can still be detected — we conjecture that autonomous systems will often exhibit this fault-masking behavior. The execution is verified against the specification, both because we don't have the tools to verify code statically, and because it provides a concrete demonstration of quality.

Transition zones are located and explored to take best advantage of limited test resources: That is where most faults will occur. We propose (currently speculative) techniques for locating the transition zones, and attempt to quantify that state space reduction that can be expected.

We first define autonomy, then provide some background information to justify its use. An outline of traditional spacecraft test methods is provided. We indicate the shortcomings of these methods with respect to autonomous spacecraft system, and finally we propose a comprehensive solution, covering technical and managerial issues.

What is Autonomy?

There is much room to argue the definition of autonomy. Smithers[3] undertakes an extensive analysis of the many definitions now in use. We adopt the definition that dominates within the aerospace community, provided some time ago by Turner at JPL[4]:

> *Autonomy* The attribute of a system to meet mission performance requirements without external support for a specified period of time.

The Webster definition is quite similar:

> *Autonomy* The quality or state of being self governing.

In the contemporary vernacular, the meaning of autonomy has moved away from the application of conventional control theory[5] to attitude control, to an arena of AI and the focus of this paper: various applications of on-board search and optimization[6], perhaps heuristic to minimize resource consumption, maximize science return, optimize a schedule, determine the root cause of a failure where the environment has substantial influence on the outcome of the algorithm. Though still a closed loop control system in a general sense, the major mathematical basis of these systems is discrete optimization (probably heuristic), rather than classical control theory.

It is the mathematical nature of such discrete optimization problems that they can be very sensitive to parameters in the sense that a seemingly small change in the input can cause a large and "non-linear" [1] difference in the output. For example, a change in the length or time of an event of a fraction of a percent can cause a planner algorithm to emit a very different plan. We make this distinction in order to stress the importance of state exploration.

Why Autonomy?

There are pragmatic and theoretical reasons to increase on-board spacecraft autonomy.

The primary theoretical reason is light-time communications delay: Ground control of an operation that requires tight feedback is either impossible (if real-time demands can't be met) or inefficient. You can't "joy stick" a rover on Mars like you could if it were in your backyard, nor can ground control respond to an on-board fault that requires immediate response in order to prevent damage to the mission.

Pragmatic concerns also center on communication: it's very expensive to communicate with a spacecraft in deep space; you'll always want more bandwidth than you can get; and ground controllers are expensive. It follows that autonomy will be used to improve science information density, by doing on-board targeting and "culling", and by reducing down-time caused by fault recovery. Finally, there are vast opportunities to reduce costs and improve safety if autonomy can be used to replace humans in space.

Autonomy obviously becomes more important as we reach farther into space, but can be profitably applied to Earth-orbit military and commercial spacecraft as well. Antenna time and operations overhead can both be reduced.

2 TRADITIONAL TESTING

Traditional system-level testing of spacecraft software at JPL basically confirms that each command works as expected; each requirement has been met; and that all sequences of commands that will probably be used during the mission work. This has been effective because spacecraft software has been designed to minimize subsystem interactions and sensitivity to environmental concerns, so a feature, if it works at all, will probably always work. This is the fundamental assumption upon which the soundness of the technique is based.

This technique assumes that there is a way to tell if a sequence of commands did indeed execute properly. Spacecraft flight software usually has a number of auditing mechanisms that are used to confirm that the spacecraft is operating properly during the mission. There are, for example, numerous counters that track the number of times various events have occurred. These counters are made visible in the spacecraft telemetry, and are used to check test results by comparing the various counter values with predicted values that also generated as a matter of routine during the operation of the spacecraft.

[1] Some call this behavior "chaotic". We don't, because we don't know if it meets the formal criteria defining a chaotic system.

This technique dots not scale to autononomy-rich systems for two reasons: Autonomous systems tend to have many more subsystem interactions; and they tend to be more sensitive to the environment and current state of the spacecraft. Both of these greatly increase the context sensitivity of commands, and so tend to invalidate the "if it works at all, it'll always work" premise. As a result, the confidence gained per test goes down, so more tests must be performed. But, the tests must vary the environment and system state trajectory, which is fundamentally different than what is now done. All of these things suggest that a new technique is required.

3 Testing Autonomous Software

Why Autonomous Systems are Different

Ultimately, our objective in testing is to improve the expected science return, and so minimize the incremental cost of science data. Science return earl become non-optimal in many ways, including:

- 1,oss of spacecraft.
- Missal targetting opportunity.
- Sill)-ol)tirtlal targetting.
- 1,0ss of acquired data.
- Missal downlink opportunity.
- Inefficient use of downlink bandwidth.
- Inoperable instrument.
- Inefficient use of spacecraft resources.
- Untimely change in spacecraft configuration.
- Rendezvous or pointing maneuvers too complicated.

These risks arc present in all science spacecraft. The interesting thing about autonomous systems is that many things that were traditionally under ground control and were thus the responsibility of human controllers become the responsibility of the autonomy system. This must lead to additional test obligations.

A major goal of the testing we advocate is to demonstrate that the spacecraft is in some sense robust in the face of "routine" failures, and that the autonomy component in particular makes good use of resources in other words, that the autonomy components work as intended. We anticipate that our methods will not only demonstrate the properties outlined above, if they are present, but will also greatly aid in providing the properties, by helping the developers locate weaknesses in the system so that they might be removed. Once these things are done, the testing has met its objective of increasing expected science return.

Premature spacecraft failure is the biggest threat to science return in terms of 10ss potential, so a significant part of autonomy software tries to protect the spacecraft against on-board failures and self-dd ructive commanding[2]. Unfortunately, since an autonomous system by definition has (within design constraints, of course) substantial control over its own fate, it follows that the spacecraft is highly vulnerable to mistakes within the autonomy design and implementation, and so should be heavily exercised.

If we stipulate that the spacecraft can't be commanded to cause itself permanent harm, then the next biggest threat to science data return is to command the spacecraft to do something of low science value. This could occur, for example, if a science request conflicted with other pending requests, which would trigger on-board conflict resolution and consequent suboptimal science return. It will therefore remain important that we have a method of confirming that commands to the spacecraft will provide good science return, even if maintenance of spacecraft health is no longer a concern.

There appear to be three things that make autonomous spacecraft software "different" as compared to traditional mission-critical flight systems: The technology is more complicated and less mature; Autonomous systems tend h) have more subsystem interactions; and autonomy makes the spacecraft more perceptive of and sensitive to itself and its environment (Indeed, that's the point of autonomy!). Taken together these things constitute a "quantum leap". Sound management practice dictates that one must carefully examine all assumptions when such leaps occur.

The complexity of autonomy software, and the low maturity of autonomy technology in general, lead directly to performance uncertainty: Given the state of software development technology today, you just don't know what the system is going to (lo until you try it. You know what it's *supposed* to do, and what it has *so far* been observed to do, but that's much different than having confidence in its behavior under all likely mission scenarios. But without that confidence, you earl't rationally allow the technology to control the fate of an expensive spacecraft.

New Testing Paradigm

One way (perhaps the only way) to provide confidence in an autonomous system today is to exercise the system extensively via a large number of simulated missions, cacti proximate in some sense to the nominal mission, and confirm the correct behavior of the system over each mission[3]. Complexity is discussed further in the section on risk management.

Figure 1 outlines the system we propose. In addition,

[2] Spacecraft have always had a powerful on-board fault protection capability. Modern autonomy enables greater fault coverage and responses that are more likely to allow the mission to progress without human involvement, and thus delay, in the recovery process.

[3] Even then, one is faced with the "abstraction problem": The simulation is an abstraction of the universe, and it can be difficult to convince people that one retained all essential features of the universe in the abstraction. We've seen many lively discussions come of this situation.

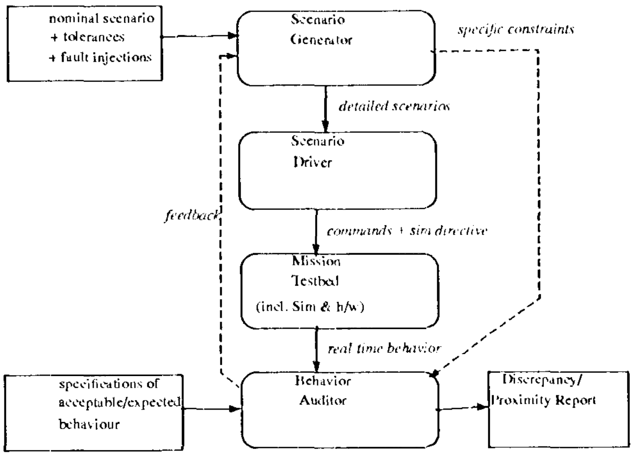

Figure 1: Overview of system

one must have a simulator of the spacecraft upon which to execute the autonomous software, as well as a simulation of the universe to stimulate the spacecraft sensors and react with the actuators. Fortunately, the High-speed Spacecraft Simulator[7] (HSS) and Dynamics Algorithms for Real-time Simulation (DARTS) dynamics simulator[8] have been available for some time. One may also wish to consider executing the autonomy subsystem within an abbreviated simulator, so the simulations execute more quickly. It might even be possible to design the autonomy system to work within a discrete event simulation, so that many, very high performance "quick look" simulations can be performed. Most of the time there isn't anything "interesting" happening insofar as the autonomy subsystem is concerned. Such arms probably won't be worth exploring in detail. It would be nice to have a way of moving quickly over them.

Our technique is based upon this process:

Define the nominal mission. The key to our approach is the execution of a large number of perturbations of and "near" the nominal mission. This requires that the nominal mission be formally[4] defined, and that there is a way to express the bounds of plausibility around the nominal mission. The issue here isn't just variance of trajectory and resource consumption, but, more interestingly, variance in the timing and occurance of fault conditions. We have done some preliminary (as yet unreported) work in this area, but as yet have no solution to offer.

Generate mutations of the mission. There are an effectively infinite number of plausible mission state trajectories. How do we generate a subset both small enough to simulate and that provides "good" coverage? We believe that a combination of manual, Monte Carlo, and feedback techniques will be necessary. Manual methods will be needed for the foreseeable future to cover areas for which we have no algorithmic approach. One may, for example, have a list of fault conditions that are particularly interesting to certain people. Monte Carlo will be used to cover spaces of more or less uniform density, for example to model resource consumption not near boundary conditions, or dynamics during the cruise phase of the mission, where nothing much interesting is happening. Feedback techniques will be necessary where the space is very large and non-uniform. For example, search and optimization-based subsystems (e.g. planners and schedulers) have regions where the response is linear-like with respect to the stimulus, and other regions where they are in transition and have highly non-linear responses. We speculate that the latter can be located and stressed automatically. Such regions will tend to be a rich source of faults, and so should be paid particular attention.

Simulate each mission. We check that the state trajectory of an execution of the system is a member of the set of all correct trajectories (generated implicitly using the formal specification of correct behavior of the system). In this step of the process, a trajectory is computed for a given mutation of the baseline mission. The mission may be executed upon a spacecraft simulator, or even a breadboard of the spacecraft. Performance is important, though, so it would be best to execute it on an abstract simulator rather than the breadboard, unless one is especially paranoid about abstraction.

Determine system behaved correctly. Our method will generate a tremendous amount of data that must be analyzed. There won't be time for manual inspection, and many faults will probably be subtle, automatically corrected by the system being tested, and thus remain unnoticed anyway. A formal specification of system behavior is created, that reflects both black-box and white-box behaviors. The latter, though a bit untraditional for use during system-level testing, will greatly increase the efficiency of the tests, because autonomous systems tend to converge after minor faults, such that the fault may not manifest itself as an externally-visible failure. However, such behavior is not cause for celebration: there was a fault, and under different circumstances it could become a big failure. Better to locate and fix such problems on the ground, than to discover them during flight! It is not easy to develop such specifications[9]. We have done preliminary work on a simple axiomatic system with temporal capabilities called TAUDIT (unpublished) that can quickly check a large volume of data against many axioms, and TSPEC (unpublished) which provides user-friendly constructs that are compiled into TAUDIT but are generally much easier for non-logicians to read and write. The temporal capability of TAUDIT isn't powerful enough to naturally express some protocols, which is nonetheless necessary because

[4] Defined such that automated analysis and manipulation is practical.

some behavior of autonomous systems is basically manifested into interleaved protocols in the messaging system. The challenge is to highlight for manual investigation messages that can't be explained. We've investigated the use of a variation of Augmented Transition Networks[10] and various state machine notations[11] for that purpose, but have not implemented anything.

Determine proximity to failure. We would like to know not just that all of the tests were passed, but by *how much*. A TAUDIT specification contains a large number of predicates. It seems to us that it should be possible to develop some notion of proximity to failure for each predicate, at least in a heuristic sense. This, in turn, could be used both to provide some measure of confidence in the tested software, and to drive the test scenarios towards additional exploration of potentially weak areas. This work has not advanced beyond a few lunchtime conversations, but does show promise.

4 Formal Specifications

A formal specification is a mathematically precise statement of how the software is expected to behave[9]. Our approach is not dependent upon the particular notation that is used, as long as it meets one criteria: We must be able to write a program that uses it ttj(:t)rl[irtll/(let]}' that a given state trajectory conforms to the specification.

For various theoretical and practical reasons it is not possible to write a specification for most software products that can classify any state trajectory as either correct or not correct. It is, however, quite practical to get very close, and that is what we advocate.

We developed a formal notation called TAUDIT that is specifically designed to specify and test transition-oriented systems. It is based upon propositional logic, but includes some temporal operators that have proven useful in practice. Figure /reffig:taudit shows part of a TAUDIT specification for a Microprocessor, which should give an idea of the notation (details are not important at this point).

Essentially, each invariant is checked whenever any of the variables upon which its value depends change in value. If the invariant is false then a diagnostic message is generated. The notation includes the usual arithmetic, logical, and relational operators, as well as user-}vrittctl functions and the special operator "prev" to access the previous value of an expression.

5 Risk Management

There is little published on testing, proving the correctness of, or identifying the risk factors within, most of the components of a contemporary autonomous system (planners, schedulers, expert systems, search engines, model-based fault detection/recovery algorithms). We have, however, identified a number of standard management risk indicators possessed by a contemporary autonomous system:

- 11's an unprecedented product.
- it's advanced software development.
- It's a real-time system.
- *11's* an embedded system.
- It's components are tightly coupled.
- It must be of the highest quality.
- It requires highly specialized software skills.
- It's probably constrained by cost and schedule.

Taken together, these clearly indicate that management rigor is required. Many of these indicators are called out in Boehm[12], which also discusses general software risk management and mitigation in some detail. It should be read by anybody undertaking the management of a significant software project. Others have IX'('II demonstrated to extend software development schedules, as evidenced in some of the COCOMO[13] schedule/workforce estimator coefficients. JPL has published a good lessons-learned[14] that came of the development of an autonomous spacecraft system, as has ESA[2]. Both will be useful to anybody undertaking a similar development effort.

```
funcdecl add8(_al, _a2 )
  (_a1+_a2)%256;

funcdecl bv(_a1)
  (_a1)?1:0;

funcdecl addcommon(_a1,_a2)
  rA <- add8(prev(rA),_a1)
  & fCY <- bv((prev(rA)+_a1)>255)
  & fS <- bv(add8(prev(rA),_a1) > 127)
  & fZ <- bv(add8(prev(rA),_a1) = 0)
  & fP <- bv(even_parity(add8(prev(rA) ,_a1)))
  & fAC <- bv((prev(rA)%16+_a1%16) > 16)
  & nc(_a2,{rA},{});

#! ADD r           5-6
invariant ADDr     op_nns(#b10000000) ->
          addcommon(rSSS ,1);
```

Figure 2: TAUDIT example

6 Summary

We must demonstrate the robustness of autonomous spacecraft before it makes sense to base the success of a mission upon such a system. Autonomy introduces factors that tend to invalidate traditional spacecraft system test methods, so we need new methods that will be effective when applied to newer spacecraft software. We propose a method based upon the execution of a large number of mutations of a nominal mission scenario and the use of automated analysis of white-box Lest results, using formal specifications of expected behavior.

The introduction of autonomy to spacecraft systems also introduces software development risk indicators either attenuated or not present in "traditional" flight software, which must be managed if a successful product is to be built.

REFERENCES

[1] S. Hedberg. AI Coming of Age: NASA uses AI for Autonomous Space Exploration. *IEEE Expert*, pages 13 15, June 1997.

[2] W. Wimmer, P. Ferri, and H. Hubner. On-board Autonomy, EURECA Experience and Requirements for Future Space Missions. *Control Engineering Practice*, 4(1 2):1715 172~), 19!)6.

[3] T. Smithers. Autonomy in Robots and Other Agents. *Brain and Cognition*, 34(1):88 106, June 1997.

[4] P.R. Turner. Autonomy and Automation for Space Station Housekeeping and Maintenance Functions. *Journal of Engineering for Industry*, 107(1):39 42, February 1985.

[5] P.J. Antsaklis. On Autonomy and Intelligence in Control. *IEEE Control Systems Magazine*, 14(3):61- 62, 1994.

[6] D.E. Bernard et al. Design of the Remote Agent Experiment for Spacecraft Autonomy. In *IEEE Aerospace Conference Proceedings*, Aspen, (k)., March 1998. Submitted to.

[7] A. Morrissett et al. Multimission High Speed Spacecraft Simulation For The Galileo and Cassini Missions. In *AIAA Computing in Aerospace Conference 9th*, S071 *Diego, CA, October* 19-2?1, *1993.* American Institute of Aeronautics and Astronautics, October 1993.

[8] J. Biesiadecki, A. Jain, and M.L. James. Advanced Simulation Environment for Autonomous Spacecraft. In *International Symposium on Artificial Intelligence Robotics and Automation in Space (i-SAIRAS97)*, Tokyo, Japan, July 1997.

[9] Formal Methods Specification and Analysis Guidebook for the Verification of Softare and Computer Systems. Technical Report NASA-GB-001 97, NASA Office of Safety and Mission Assurance, 1997.

[10] W. A. Woods. Transition network grammars for natural language analysis. *Communications of the ACM*, 13(10):591 606, October 1970.

[11] D. Harel and A. Naamad. The statemate semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293 333, October 1996.

[12] B.W. Boehm. Software risk management: Principles and practices. *IEEE Software*, 8(1):32 41, January 1991.

[13] B.W. Boehm. *Software Engineering Economics*. Prentice Hall.

[14] A.S. Aljabri, D.L. Dvorak, G. K. Man, and T. W. Starbird. Infusion of Autonomy Technologies Into Space Missions - DS1 Lessons Learned. In *IEEE Aerospace Conference Proceedings*, Aspen, Co., March 1998. Submitted to.